Toward A Learning Apprentice for Software Re-engineering

Michael R. Lowry

Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

Smadar Kedar

Sterling Federal Systems
AI Research Branch, Mail Stop 244-17
NASA Ames Research Center
Moffett Field, CA 94035

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA–91–14

May 1991

Toward A Learning Apprentice for Software Re-engineering

Michael R. Lowry
Kestrel Institute
3260 Hillview Ave.
Palo Alto CA 94304
lowry@kestrel.edu

Smadar T. Kedar
Sterling Federal Systems
NASA Ames Research Center
Moffett Field CA 94035
kedar@ptolemy.arc.nasa.gov

Abstract

Most programmers spend their time in maintenance activities, such as porting programs. Existing tools provide only limited help because of their lack of flexibility. This paper describes research toward an interactive software re-engineering learning apprentice that learns translation rules for porting programs between different languages and different hardware platforms. The programmer shows examples of corresponding code fragments, and the learning apprentice generalizes these examples. First the learning apprentice syntactically generalizes the correspondence, then semantically verifies the correspondence, and then generalizes the verification proof to derive a translation rule with preconditions using explanation-based generalization. When a verification does not succeed, the learning apprentice determines sufficient conditions which can be incorporated into the translated program.

This paper appears in *The Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91) Workshop on Automating Software Design*, held in Anaheim, CA, July, 1991.

Toward A Learning Apprentice for Software Re-engineering

Michael R. Lowry
Kestrel Institute
3260 Hillview Ave.
Palo Alto CA 94304
lowry@kestrel.edu

Smadar T. Kedar
Sterling Federal Systems
NASA Ames Research Center
Moffett Field CA 94035
kedar@ptolemy.arc.nasa.gov

Abstract

Most programmers spend their time in maintenance activities, such as porting programs. Existing tools provide only limited help because of their lack of flexibility. This paper describes research toward an interactive software re-engineering learning apprentice that learns translation rules for porting programs between different languages and different hardware platforms. The programmer shows examples of corresponding code fragments, and the learning apprentice generalizes these examples. First the learning apprentice syntactically generalizes the correspondence, then semantically verifies the correspondence, and then generalizes the verification proof to derive a translation rule with preconditions using explanation-based generalization. When a verification does not succeed, the learning apprentice determines sufficient conditions which can be incorporated into the translated program.

1 Introduction and Motivation

Most programmers spend their time maintaining programs, including porting existing programs to new languages and new hardware platforms. Although commercial off-the-shelf translators are available, they are seldom used. Such translators treat the target language as a machine language, making the resulting code unreadable. It is also difficult to specialize these translators to different programming language dialects. Thus although these translators sometimes satisfy the minimal functional goals of porting – enabling code to be run on a new platform– they do not satisfy the non-functional goals such as maintainability which are usually the primary motivation for porting code. For example, despite the commercial availability of Fortran to C translators, one NASA site has contracted 40 maintenance programmers to manually port existing flight control software in Fortran on IBM mainframes to C on Unix workstations.

Because re-engineering is usually motivated by non-functional goals, we believe that an interactive re-engineering system is preferable to a completely automatic re-engineering system. In particular, a re-engineering system should assist a maintenance programmer who is porting a program by ensuring that functional goals are satisfied, such as target program correctness, while providing implementation freedom in meeting non-functional goals. Non-functional goals are difficult to formalize and are often best conveyed interactively by example. Ideally, such an interactive system would incorporate a knowledge acquisition capability, so that the maintenance programmer can simply provide examples of how code fragments should be translated, and the system will generalize these examples into translation rules. This type of interactive aid to knowledge acquisition and generalization is known as a learning apprentice system (Mitchell, *et. al.*, 1985) (Smith, *et. al.*, 1985). These learning apprentice systems can acquire and generalize new knowledge by observing and analyzing specific examples provided by a user. This paper describes research toward such a learning apprentice for software re-engineering. A prototype system that syntactically generalizes examples of translated code fragments, the Generic Translator, has already been implemented. The main portion of this paper describes an approach to extending the Generic Translator to a Learning Translator (LTran) that semantically generalizes examples of translated code fragments.

As a motivating example, consider the problem of porting a Fortran program to a C program. There are many syntactic and semantic differences between Fortran and C, such as: Fortran arrays have a default lower bound of 1 while C arrays have a default lower bound of 0; Fortran arrays are stored in column order while C arrays are stored in row order. The maintenance programmer needs to consider these sometimes subtle differences while at the same time addressing performance, maintainability, stylistic and other non-functional goals. In one context the programmer may want to translate a Fortran array to a C array by decrementing each index of the Fortran array by 1. This conserves space but can make the code more difficult to read. Also, manually carrying out this translation for all the array operations usually introduces errors. In another context, the programmer may want each index of the C array to correspond directly to that of the Fortran array. This choice can make the resulting code easier to read and maintain, but also introduces a subtle difference in the behavior of the Fortran array operations and the corresponding C

operations. In our approach, LTran verifies that a particular example of decrementing the array index is semantically valid, and then generalizes this example into translation rules. These rules would then be applied automatically, eliminating manual error. For the second option of not decrementing, LTran finds sufficient conditions for the translation to be valid, and then would interact with the user to determine how to achieve these conditions; for example, inserting a conditional branch on these conditions into the translated program.

This paper first describes the implemented Generic Translator. It then describes the mathematics for verifying translations and outlines the extended approach, the Learning Translator. The Learning Translator uses explanation-based generalization (EBG) (Mitchell, *et. al.*, 1986) to verify and generalize example translations provided by the user. The paper then shows two detailed examples, the first illustrating the acquisition of general translation rules that follow directly from the syntactic translation. The second example illustrates the acquisition of general translation rules when additional sufficient conditions need to be added in order for the syntactic translation to be semantically correct.

2 The Generic Translator

The goal of the Generic Translator is to be an easily programmable translator between different programming languages and different hardware platforms. The latest version of the Generic Translator can be programmed by example, but the examples are only syntactically generalized. The Learning Translator will extend this capability by semantically generalizing examples. The first version of the Generic Translator was developed by Philip Newcombe as a tool for porting about two dozen programs from Basic to Fortran. This version was written in Basic; translation rules were defined through patterns on text strings. The effort in building the Generic Translator was more than paid back in the vast reduction of manual effort to port just those two dozen programs. In order to disambiguate program structure in the source text, rules were fairly specific and consequently numerous. Rules could be defined 'on the fly' during translation, so that when the translator was unable to match program text to rules in its libraries it could notify the user. The user would then type in a rule and the Generic Translator would continue.

The second version of the Generic Translator was developed by Lowry. The objective was to provide a graphical user interface for translating programs, provide a friendly user interface for defining translation rules, and to use abstract syntax trees as the underlying representation. Abstract syntax trees (AST) are essentially parsed programs in which the syntactic structure of the program is explicitly represented in a tree. ASTs provide a higher-level and syntactically unambiguous representation for program structure, so rules can be more general. This second version was written in Refine™, a knowledge-based environment for program derivation and analysis that includes parser/printer generators, an object-oriented knowledge base, support for translation rules, and a programmable graphical user interface. The graphical interface allows the user to select subtrees in the AST representation by mousing on the source text in a display window. Grammars for Fortran and C were provided by Reasoning Systems, Inc. Refine compiles these grammars into parsers and printers that translate between the source text representation and the AST representation (Burson, *et. al.*, 1990).

The Generic Translator enables the user to parse source programs, apply translation rules automatically or interactively, and write the translated programs into a text file. The translation rules are context-dependent and by default are applied top down through recursive descent. At intermediate stages of translation, the target program consists of mixed syntax from the source language and the target language. If the set of translation rules is not complete, then the resulting target program will contain fragments of the source language. The user can then either finish the translation by hand or interactively define new translation rules.

The Generic Translator has two modes for interactively acquiring new translation rules. In both modes, a user selects a code fragment in the source language, which the system generalizes by replacing the topmost subtrees in the AST representation with pattern variables. A facility for interactive generalization/specialization of the source code fragment was designed and partially implemented. This allows a user the option of abstracting the subtrees at any level of the AST hierarchy, instead of just the top level. In the first mode for rule acquisition, the system displays the left hand side pattern obtained by generalizing the source code fragment, and then the user types in the right hand side pattern using a mixture of the target language syntax and Refine's pattern language syntax. This mode requires the user to be familiar with Refine's notation. In the second mode for rule acquisition, the user loads in the corresponding program in the target language into another window, and then defines the right hand side of the translation rule by selecting and generalizing a target language code fragment. After the right hand size is generalized, the user then indicates the correspondence between the pattern variables in the left hand side and right hand side. This mode is suitable for users unfamiliar with Refine's notation and the concepts underlying its operation. A simple example of an actual translation rule acquired interactively in this second mode is the following:

```
rule LE-EXPRESSION-8 (var A)
    A = '##r PATTERN-FORTRAN-77
          @U .le. @V'
    ⟹
    A = '##r C #E
          @U ≤ @V'
```

This rule converts less-than-or-equal expressions in Fortran-77 into the corresponding expressions in C. The first line is the declaration of the rule, whose name is derived automatically from the name attribute of the root of the left hand side. The second line is Refine's notation for specifying the Fortran-77 grammar with patterns. The third line is the left hand side pattern, which in this case matches a Fortran-77 arithmetic comparator

expression for less-than-or-equal; @U and @V are pattern variables. The fourth line is Refine's notation for specifying an expression in the C grammar. The fifth line is the corresponding right hand side in the C pattern language. After acquiring a rule, the Generic Translator compiles it and adds it to a library so it may be applied in subsequent translations.

3 The Learning Translator

Although the Generic Translator can interactively learn syntactic translation rules, it does not provide any assistance with the semantics of translation. In this section we describe the design of an extension to the Generic Translator, the Learning Translator, which acquires general translation rules that are semantically valid with the same user interface as the Generic Translator. The Learning Translator (LTran) replaces the Generic Translator's syntactic generalization method with a more powerful semantic generalization method.

The method which enables LTran to generalize from the analysis of just one example is known as explanation-based generalization, or EBG. EBG's ability to generalize from one example is based on its ability to generalize a proof that the example is valid. It generalizes the proof for an example by retaining only the features of the example needed for the proof to hold. The other features of the example are dropped, since they are incidental to the proof. EBG can be cast as a variant on resolution theorem-proving for horn clause logic. This variant can be implemented as a modification of the PROLOG interpreter, called PROLOG-EBG (Kedar-Cabelli & McCarty, 1987). PROLOG-EBG first constructs a proof for the example. It then constructs a generalized proof, mirroring the specific proof, yet retaining only variable bindings needed for the proof, and none that were introduced by the example alone.

The Learning Translator uses the following method for acquiring new translation rules:

1. **Syntactic Translation:** As in the Generic Translator, the user indicates correspondences from a source code fragment to a target code fragment. From these correspondences, syntactic translation rules are derived as described in the previous section.

2. **Operational Semantics:** Theories for the operational semantics of these code fragments are derived by instantiating generic axioms for the operations in the code fragments. These generic axioms are language specific, and are defined by an expert.

3. **Verification:** LTran attempts to verify that the syntactic translation is semantically valid, i.e. that the translation rules define a *theory interpretation* from the theory for the source code fragment to the theory for the target code fragment.

4. **Sufficient Condition Generation:** If LTran cannot semantically verify the syntactic translation as is, it derives sufficient conditions for the theory interpretation to hold.

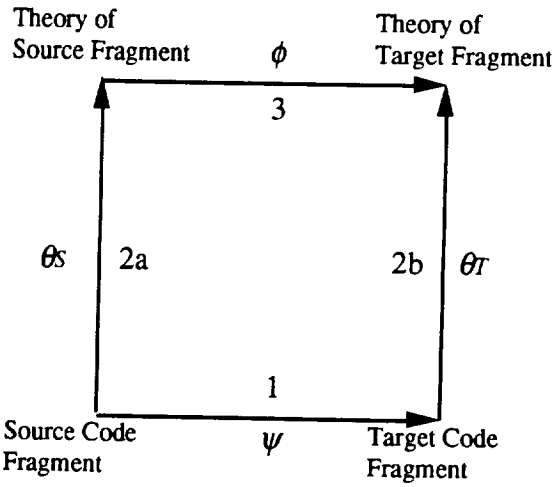

Figure 1: Commutative Diagram for Learning Translator

5. **Generalization:** Using EBG, the proofs and the sufficient conditions are generalized into associated general preconditions for applying the syntactic translation rules.

Figure 1, labeled with the first 3 steps, is the commutative diagram which is the basis for this method. This diagram can be summarized by the slogan 'The meaning of the translation implies the translation of the meaning'[1].

LTran verifies that the syntactic translation ψ is semantically valid by proving that it defines a theory interpretation from the source to target theories. A theory interpretation consists of a source theory, a target theory, and a map ϕ from the syntax of the source theory to the syntax of the target theory. A theory interpretation is valid precisely when each of the axioms of the source theory, translated under ϕ, are derivable from the target theory. Theory interpretations can be used for refining specifications to implemented code (Blaine & Goldberg, 1991); LTran essentially treats the source as a specification and the target as the code.

LTran derives the map ϕ from the syntactic translation ψ defined by the user, the map θ_S from the source code language to the language of the source theory, and the map θ_T from the target code language to the language of the target theory. If SL is the source language, then map ϕ is defined as follows:

$$\phi(\theta_S(SL)) = \theta_T(\psi(SL))$$

LTran derives the theory for the source code fragment (call it FC) by instantiating generic axioms associated with the map θ_S. Similarly, LTran derives the theory for the translated source code fragment, $\psi(FC)$, by instantiating generic axioms associated with the map θ_T. LTran verifies the theory interpretation by translating, using ϕ, each axiom in the source theory into the language of the

[1] Richard Jullig, personal communication.

target theory and proving that each translated axiom is derivable from the target theory. Thus the axioms derived by assigning an operational semantics to the source code fragment (step 2a) and then translating to the language of the target code (step 3) must be derivable from the axioms derived by first translating the source code into the target code language (step 1) and then assigning operational semantics to the target code (step 2b). Thus the following relationship must hold, where FC is the original source code fragment:

$$\theta_T(\psi(FC)) \vdash \phi(\theta_S(FC))$$

A proof of this relationship for a particular code fragment FC is generalized using EBG (step 5) to a proof of this relationship for the syntactic generalization defined by the Generic Translator.

When the relationship above cannot be proved for a particular code fragment, LTran takes the unresolved clauses in a resolution refutation proof and derives sufficient conditions (SC) for the proof to succeed (step 4):

$$SC \wedge \theta_T(\psi(FC)) \vdash \phi(\theta_S(FC))$$

These sufficient conditions are likewise generalized by EBG (step 5).

In our approach, axioms that define the operational semantics of a programming language are defined by experts. These axioms do not have to be complete; LTran will do its verifications and generalization with whatever axioms are provided. (This flexibility to deal with partial theories is crucial to the success of this approach, since automatically proving implementation correctness is a difficult problem.)

In the examples which follow, the axioms that define Fortran array operations and C array operations are slightly different instantiations of the following abstract data type (ADT) for single dimensional arrays. The first part of this ADT definition is a declaration of the types of objects and the operations on these objects. The types of objects are arrays, indices into arrays, and values stored in arrays. The indices range over the integers. The array operations are: array declaration (which defines upper and lower bounds), 'get' and 'put' operations to retrieve and insert values from an array, and auxiliary operations to retrieve the lower and upper bounds of an array. In the examples which follow we are not concerned with the types of values stored in the arrays; a more complete ADT for arrays would include the types of values as a parameter to the theory. The second part of this ADT definition are the axioms, which include error conditions when array upper bounds and lower bounds are violated.

Declare ARRAY
Types : $array, index, value$
Operations : $get, put, declare, lb, ub$
get : $array \times index \rightarrow value$
put : $array \times index \times value \rightarrow array$
$declare$: $array \times index \times index \rightarrow array$
lb : $array \rightarrow index$
ub : $array \rightarrow index$
Axioms :

$lb(declare(A,x,y)) = x$
$ub(declare(A,x,y)) = y$
$e < lb(A) \Rightarrow put(A,e,x) = error$
$e > ub(A) \Rightarrow put(A,e,x) = error$
* $e < lb(A) \Rightarrow get(A,e) = error$
$e > ub(A) \Rightarrow get(A,e) = error$
$e \geq lb(A) \wedge e \leq ub(A) \wedge e = k$
$\Rightarrow get(put(A,k,x),e) = x$
$e \geq lb(A) \wedge e \leq ub(A) \wedge e \neq k$
$\Rightarrow get(put(A,k,x),e) = get(A,e)$

The starred axiom will be the focus in the following examples. This axiom denotes an 'out of bounds' error condition that arises when an array access is attempted with an index that is less than the lower bound of the array. The application of LTran's verification and generalization method to this axiom is representative of application to other axioms. In the following examples, the semantics for operations on data types is specified through a map θ, which maps operations in the programming language onto operations in ADTs. This map is defined by an expert for each language; the maintenance programmer using LTran does not need to know about the details of this map nor about the details of the underlying ADTs. Below we define these maps for Fortran (θ_F) and C (θ_C) array operations on single-dimensional arrays. Note that the default lower bound for Fortran arrays is 1, while for C the default lower bound is 0. Thus array declarations without a lower bound specification will result in different lower bounds in the two languages. Also, the upper bound of a Fortran array is the size of the array, while for C the upper bound is one less than the size of the array.

Map for Fortran, with partially evaluated axioms for Fortran array access:

$$\theta_F(Real\ A(M)) = declare_F(A,1,M)$$
$$\theta_F(A(e)) = get_F(A,\theta_F(e))$$

Partially evaluated axioms:

$\theta_F(e) < 1 \Rightarrow get_F(A,\theta_F(e)) = error$
$\theta_F(e) > M \Rightarrow get_F(A,\theta_F(e)) = error$
$\theta_F(e) \geq 1 \wedge \theta_F(e) \leq M \wedge \theta_F(e) = k$
$\Rightarrow get_F(put_F(A,k,x),\theta_F(e)) = x$
$\theta_F(e) \geq 1 \wedge \theta_F(e) \leq M \wedge \theta_F(e) \neq k$
$\Rightarrow get_F(put_F(A,k,x),\theta_F(e)) = get_F(A,\theta_F(e))$

Map for C, with partially evaluated axioms for C array access:

$$\theta_C(Float\ A[M]) = declare_C(A,0,M-1)$$
$$\theta_C(A[e]) = get_C(A,\theta_C(e))$$

Partially evaluated axioms:

$\theta_C(e) < 0 \Rightarrow get_C(A,\theta_C(e)) = error$
$\theta_C(e) > M-1 \Rightarrow get_C(A,\theta_C(e)) = error$
$\theta_C(e) \geq 0 \wedge \theta_C(e) \leq M-1 \wedge \theta_C(e) = k$
$\Rightarrow get_C(put_C(A,k,x),\theta_C(e)) = x$
$\theta_C(e) \geq 0 \wedge \theta_C(e) \leq M-1 \wedge \theta_C(e) \neq k$
$\Rightarrow get_C(put_C(A,k,x),\theta_C(e)) = get_C(A,\theta_C(e))$

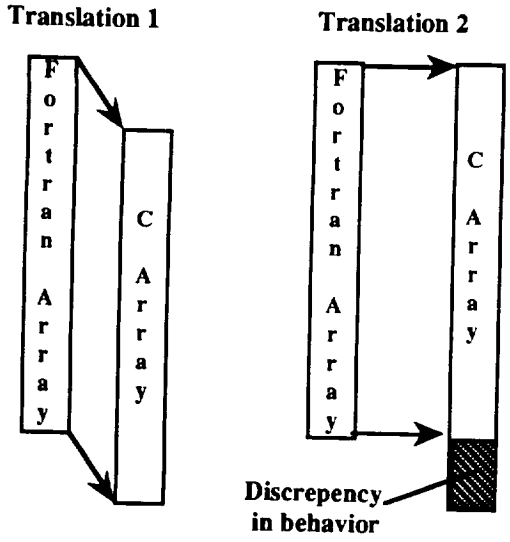

Figure 2: Two Translations of Fortran to C Arrays

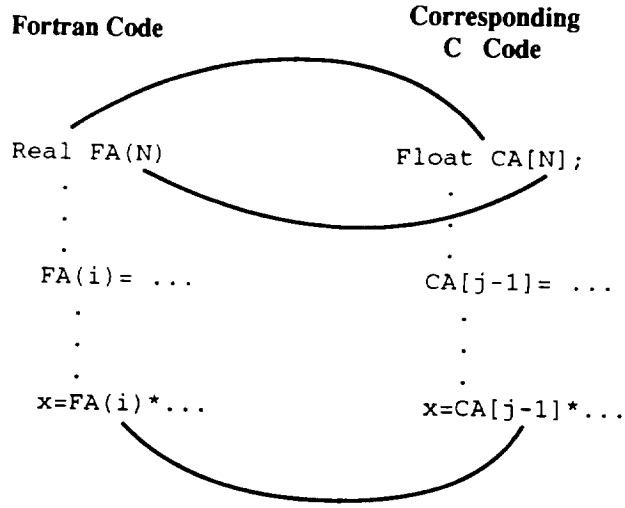

Figure 3: Translation 1

In the next sections we elaborate the steps of LTran's method through two detailed examples of different ways of translating Fortran arrays into C arrays. We consider two possible translations: In the first, the Fortran array indices are decremented by 1 in the C implementation. Because the default lower bound for C arrays is 1 less than the default lower bound for Fortran arrays, this is a correct implementation. In the second translation, the array indices are identical between the Fortran and C arrays. This translation usually results in clearer and more readable C code, however it introduces a subtle difference in behavior: when an array index of 0 is used for a Fortran array the result is an 'array out of bounds' error, while for the corresponding C array the result is not an error (the 0th entry of the C array will be accessed). These two translations are illustrated in Figure 2.

In the first translation, the system will verify that the correspondence defines a correct implementation, in the second case the system will derive a sufficient condition, namely that the array index not be 0, for the implementation to be correct.

These two translations can coexist, as long as they are not applied to the same arrays. Since the ADT axioms for the array operations 'declare', 'get', and 'put' are interdependent, the translation of the operations on each separate Fortran array must be consistent. One advantage of our ADT approach to axiomatizing the semantics of operations in a programming language is that it makes these interdependencies explicit.

Some of the operations in the source theory are auxiliary: they do not explicitly correspond to any operation in the source programming language. In the following array examples, the lower bound and upper bound operations are auxiliary: they help to define the semantics of the main array operations, but are not themselves part of the syntactic translation. These auxiliary operations are evaluated before translation in the verification proofs for the theory interpretation. In the generalizations of these proofs, terms headed by these auxiliary operations are essentially treated as constants. Further elaboration of the treatment of auxiliary operations with theory interpretations can be found in (Goguen & Meseguer, 1982) and (Lowry, 1990).

4 Translation 1: Decrement indices from Fortran to C Arrays

In this first translation, the user would like the index of the C array to be one less than the corresponding index for the Fortran array; in effect the whole array is shifted down by one. As long as the translation is consistent with respect to decrementing the array index across the 'declare', 'get', and 'put' operations, then the operational semantics of the Fortran code will be correctly implemented by the translation into C code. The user specifies this translation by giving examples of Fortran and C code, and the correspondences between them, as shown in Figure 3.

4.1 Step 1: Syntactic Translation

These examples are syntactically generalized using the method implemented in the Generic Translator to the following syntactic translation ψ_1 from Fortran to C. ψ_1 is actually defined on the underlying abstract syntax trees, but we define it here on text strings for presentation purposes:

$$\psi_1(Real\ A(N)) = Float\ \psi_1(A)[N]$$
$$\psi_1(A(e)) = \psi_1(A)[\psi_1(e) - 1]$$

These translations and θ_F, θ_C are used to derive the map ϕ_1:

$$\phi_1(declare_F(A, 1, N)) = declare_C(\phi_1(A), 0, N - 1)$$
$$\phi_1(get_F(A, e)) = get_C(\phi_1(A), \phi_1(e) - 1)$$

4.2 Step 2: Operational Semantics

Next, the operational semantics for these code fragments are derived through θ_F and θ_C. The axioms for the Fortran array access operation, get_F, are instantiated for this particular source code as follows:

$$
\begin{array}{l}
i < 1 \Rightarrow get_F(FA, i) = error \\
i > N \Rightarrow get_F(FA, i) = error \\
i \geq 1 \wedge i \leq N \wedge i = k \\
\quad \Rightarrow get_F(put_F(FA, k, x), i) = x \\
i \geq 1 \wedge i \leq N \wedge i \neq k \\
\quad \Rightarrow get_F(put_F(FA, k, x), i) = get_F(FA, i)
\end{array}
$$

For the corresponding C code, the axioms for C array access get_C, are instantiated as follows:

$$
\begin{array}{l}
j - 1 < 0 \Rightarrow get_C(CA, j - 1) = error \\
j - 1 > N - 1 \Rightarrow get_C(CA, j - 1) = error \\
j - 1 \geq 0 \wedge j - 1 \leq N - 1 \wedge j - 1 = k \\
\quad \Rightarrow get_C(put_C(CA, k, x), j - 1) = x \\
j - 1 \geq 0 \wedge j - 1 \leq N - 1 \wedge j - 1 \neq k \\
\quad \Rightarrow get_C(put_C(CA, k, x), j - 1) = get_C(CA, j - 1)
\end{array}
$$

4.3 Step 3: Verification

LTran first translates all the Fortran axioms to C using the map ϕ_1. LTran then attempts to prove that these translated Fortran axioms are derivable from the C theory. We focus on the lower bound axiom for the Fortran get_F operation. We illustrate the proof as it would be performed using a PROLOG resolution refutation theorem prover.

The instantiated Fortran axiom for accessing an index less than the lower bound is:

$$i < 1 \Rightarrow get_F(FA, i) = \ error$$

Given map ϕ_1, this Fortran axiom is translated to the language for the theory of the C operations (this is part of $\phi_1(\theta_F(FC))$).

$$\phi_1(i) < 1 \Rightarrow get_C(\phi_1(FA), \phi_1(i) - 1) = error$$

After the map ϕ_1 is recursively evaluated:

$$j < 1 \Rightarrow get_C(CA, j - 1) = error$$

The next step attempts to prove that the translated Fortran axiom is derivable from the C axioms, which include a similar out of bounds axiom (part of $\theta_C(\psi_1(FC))$).

$$j - 1 < 0 \Rightarrow get_C(CA, j - 1) = error$$

LTran needs to establish $\theta_C(\psi_1(FC)) \vdash \phi_1(\theta_F(FC))$, i.e. for this particular Fortran axiom (where the ellipse represents the rest of the axioms for C):

$$
\begin{array}{l}
[j - 1 < 0 \Rightarrow get_C(CA, j - 1) = error] \wedge \cdots \wedge \\
\vdash \\
[j < 1 \Rightarrow get_C(CA, j - 1) = error]
\end{array}
$$

For resolution refutation with arithmetic constraint reasoning, the goal is negated and the empty clause is derived in 7 steps. (We replace '$get_C(CA, j-1) = error$' with 'exp' for readability):

1. $[j - 1 < 0 \Rightarrow exp] \wedge \neg[j < 1 \Rightarrow exp]$
2. $[\neg(j - 1) < 0 \vee exp] \wedge \neg[\neg j < 1 \vee exp]$
3. $[\neg(j - 1) < 0 \vee exp] \wedge [j < 1 \wedge \neg exp]$
4. $[\neg(j - 1) < 0 \wedge j < 1 \wedge \neg exp] \vee$
   $[exp \wedge j < 1 \wedge \neg exp]$
5. $[\neg(j - 1) < 0 \wedge j < 1 \wedge \neg exp]$
6. $[\neg j < 1 \wedge j < 1 \wedge \neg exp]$
7. $[]$

4.4 Step 5: Generalization

The proof generalized by EBG drops any bindings introduced by the specific example. The only bindings retained are those used by unifications in the proof. Note that, interestingly, the general proof cannot mirror the specific proof exactly. In particular, in the general case the last step of the proof does not go through directly, because it depends on the value of the decrement being precisely the difference between the Fortran lower bound and the C lower bound. When the index is *between* the lower bounds of the two languages, one language would give an 'out of bounds' error, the other will not, and therefore the operational semantics of the arrays will differ. Since this would not be a valid implementation, this needs to be avoided. Thus even though in the specific case no sufficient conditions were needed, the general proof requires derived sufficient conditions, namely that only when the index into the array is less than the lower bound for the C array or greater than or equal to the lower bound for the Fortran array will the behavior of the two arrays be the same. Similar restrictions are derived by generalizing the proofs for the other axioms, which will result in sufficient conditions for the other cases of bounds discrepencies between Fortran and this C translation. These derived sufficient conditions can then either be used as a precondition for applying the translation or be used to insert conditional branches into the target code that cover those cases where the operational semantics would otherwise differ.

The general Fortran axiom is:

$$e < lb_F(A) \Rightarrow get_F(A, e) = \ error$$

Given map ϕ_1, this general axiom is translated as follows. Note that for this general case $\phi_1(e)$, $lb_F(A)$, and $\phi_1(A)$ are not evaluated:

$$\phi_1(e) < lb_F(A) \Rightarrow get_C(\phi_1(A), \phi_1(e) - 1) = error$$

The general C axioms include the similar out of bounds axiom.

$$
\begin{array}{l}
\phi_1(e) - 1 < lb_C(\phi_1(A)) \\
\quad \Rightarrow get_C(\phi_1(A), \phi_1(e) - 1) = error
\end{array}
$$

The general proof attemps to mirror the specific proof, yet differs at the last step. LTran needs to show $\theta_C(\psi_1(FC)) \vdash \phi_1(\theta_F(FC))$, which includes the following (where the ellipse represents the rest of the axioms for C):

$$[\phi_1(e) - 1 < lb_C(\phi_1(A)) \Rightarrow get_C(\phi_1(A), \phi_1(e) - 1) = error] \wedge \ldots \wedge$$
$$\vdash$$
$$[\phi_1(e) < lb_F(A) \Rightarrow get_C(\phi_1(A), \phi_1(e) - 1) = error]$$

For resolution refutation, the goal is negated and an attempt is made to derive the empty clause in 6 steps. (We replace '$get_C(\phi_1(A), \phi_1(e) - 1) = error$' with 'exp' for readability).

1. $[\phi_1(e) - 1 < lb_C(\phi_1(A)) \Rightarrow exp]$
   $\wedge\neg[\phi_1(e) < lb_F(A) \Rightarrow exp]$
2. $[\neg(\phi_1(e) - 1 < lb_C(\phi_1(A))) \vee exp]$
   $\wedge\neg[\neg(\phi_1(e) < lb_F(A)) \vee exp]$
3. $[\neg(\phi_1(e) - 1 < lb_C(\phi_1(A))) \vee exp]$
   $\wedge[(\phi_1(e) < lb_F(A)) \wedge \neg exp]$
4. $[\neg(\phi_1(e) - 1 < lb_C(\phi_1(A))) \wedge (\phi_1(e) < lb_F(A))$
   $\wedge\neg exp] \vee [exp \wedge (\phi_1(e) < lb_F(A)) \wedge \neg exp]$
5. $[\neg(\phi_1(e) - 1 < lb_C(\phi_1(A))) \wedge (\phi_1(e) < lb_F(A))$
   $\wedge\neg exp]$

This will resolve to the empty clause when any subset of its conjuncts are false. Therefore, a derived sufficient condition for this proof to succeed is:

6. $[(\phi_1(e) - 1 < lb_C(\phi_1(A))) \vee (\phi_1(e) \geq lb_F(A))]$

To reiterate, this states that sufficient conditions for this translation of Fortran to C arrays to be semantically valid (i.e. for both Fortran and C arrays to exhibit the same behavior), the decremented index has to be less than the lower bound of the C array (where both languages would give an 'out of bounds' error) or the original index is greater than or equal to the lower bound of the Fortran array (where neither language will give an 'out of bounds' error). Reformulated as an implication, if the original index is less than the lower bound of the Fortran array, then the decremented index has to be less than the lower bound of the C array.

5 Translation 2: Identical indices for Fortran and C Arrays

In the second translation the user would like the index of the C array to range over the same values as the Fortran array. This translation introduces a subtle difference in behavior: when an array index of 0 is used in a Fortran array access, the result is an 'out of bounds' error, while for the corresponding C array the result is not an error. Thus for this to be a semantically valid translation, the array index may not be 0. The user specifies this translation by giving examples of Fortran and C code, and the correspondences between them, as shown in Figure 4.

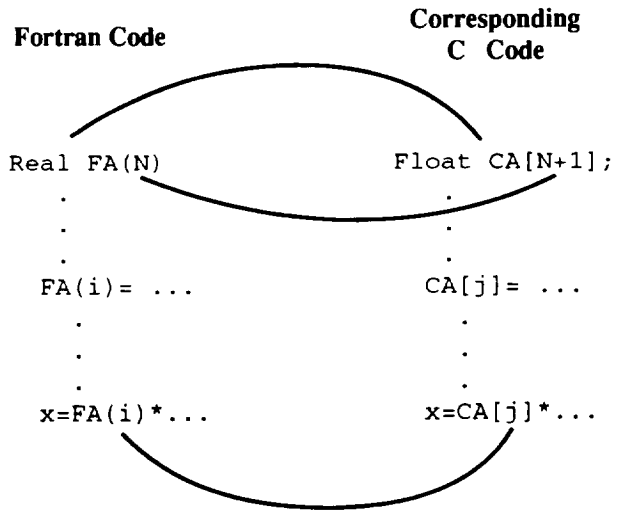

Figure 4: Translation 2

5.1 Step 1: Syntactic Translation

These examples are syntactically generalized using the method implemented in the Generic Translator to the following syntactic map, ψ_2, which is actually defined on the underlying abstract syntax trees but which we define here on text strings for presentation purposes:

$$\psi_2(Real\ A(N)) = Float\ \psi_2(A)[N+1]$$
$$\psi_2(A(e)) = \psi_2(A)[\phi_2(e)]$$

These translations and θ_F, θ_C are used to derive the map ϕ_2 :

$$\phi_2(declare_F(A, 1, N)) = declare_C(\phi_2(A), 0, N)$$
$$\phi_2(get_F(A, e)) = get_C(\phi_2(A), \phi_2(e))$$

5.2 Step 2: Operational Semantics

Next, the operational semantics for the behavior of these code fragments are derived through θ_F and θ_C.

The axioms for the Fortran array access operation, get_F, are instantiated for this particular source code fragment as follows:

$$i < 1 \Rightarrow get_F(FA, i) = error$$
$$i > N \Rightarrow get_F(FA, i) = error$$
$$i \geq 1 \wedge i \leq N \wedge i = k \Rightarrow get_F(put_F(FA, k, x), i) = x$$
$$i \geq 1 \wedge i \leq N \wedge i \neq k \Rightarrow get_F(put_F(FA, k, x), i) = get_F(FA, i)$$

For the corresponding target code fragment, the axioms for C array access are instantiated as follows:

$$j < 0 \Rightarrow get_C(CA, j) = error$$
$$j > N \Rightarrow get_C(CA, j) = error$$
$$j \geq 0 \wedge j \leq N \wedge j = k \Rightarrow get_C(put_C(CA, k, x), j) = x$$
$$j \geq 0 \wedge j \leq N \wedge j \neq k \Rightarrow get_C(put_C(CA, k, x), j) = get_C(CA, j)$$

5.3 Step 3,4: Verification and Sufficient Condition Generation

LTran attempts to verify that the syntactic translation is semantically valid by proving that there is a theory interpretation from the source to target. In particular, it attempts to demonstrate that translating the index identically between Fortran and C results in a correct implementation. LTran is unable to complete the proof. Instead it returns sufficient conditions required to complete this proof, namely that the index not be zero.

The instantiated Fortran axiom for accessing an index less than the lower bound is:

$$i < 1 \Rightarrow get_F(FA, i) = error$$

Given map ϕ_2, this Fortran axiom is translated to the language for the theory of the C operations (this is part of $\phi_2(\theta_F(FC))$):

$$\phi_2(i) < 1 \Rightarrow get_C(\phi_2(FA), \phi_2(i)) = error$$

After the map ϕ_2 is recursively evaluated:

$$j < 1 \Rightarrow get_C(CA, j) = error$$

The next step attempts to prove that the translated Fortran axiom is derivable from the C axioms, which include a similar out of bounds axiom (part of $\theta_C(\psi_1(FC))$).

$$j < 0 \Rightarrow get_C(CA, j) = error$$

LTran attempts to establish $\theta_C(\psi_2(FC)) \vdash \phi_2(\theta_F(FC))$, i.e. for this particular Fortran axiom (where the ellipse represents the rest of the axioms for C):

$$[j < 0 \Rightarrow get_C(CA, j) = error] \wedge \cdots \wedge$$
$$\vdash$$
$$[j < 1 \Rightarrow get_C(CA, j) = error]$$

For resolution refutation with arithmetic constraint reasoning, the goal is negated, and an attempt is made to derive the empty clause. (We replace '$get_C(CA, j) = error$' with 'exp' for readability):

1. $[j < 0 \Rightarrow exp] \wedge \neg[j < 1 \Rightarrow exp]$
2. $[\neg j < 0 \vee exp] \wedge \neg[\neg j < 1 \vee exp]$
3. $[\neg j < 0 \vee exp] \wedge [j < 1 \wedge \neg exp]$
4. $[\neg j < 0 \wedge j < 1 \wedge \neg exp] \vee [exp \wedge j < 1 \wedge \neg exp]$
5. $[\neg j < 0 \wedge j < 1 \wedge \neg exp]$
6. $[j = 0 \wedge \neg exp]$

This will resolve to the empty clause when any subset of its conjuncts are false. Therefore, a derived sufficient condition for this proof to succeed is:

$$[j \neq 0]$$

This states that in order for this translation to be sematically valid, the index of the C array cannot be 0.

5.4 Step 5: Generalization

This proof can be generalized by dropping any bindings introduced by the specific example not needed for the proof. LTran generalizes this proof to any translation from Fortran to C in which the array indices have the same value. The general condition under which such a translation is semantically valid is if the index into the translated Fortran array is never between the lower bounds of the Fortran and C arrays. In the case when the lower bounds of Fortran and C default to 1 and 0, respectively, the array index should not be 0. When the index is between the lower bounds of the two languages, one language would give an 'out of bounds' error, the other will not, and therefore the operational semantics of the arrays will differ. Since this would not be a valid implementation, this needs to be avoided. Thus the general proof requires derived sufficient conditions, namely that the index into the array be less than the lower bound for the C array or greater than or equal to the lower bound for the Fortran array. Similar restrictions are derived by generalizing the proofs for the other axioms. These derived sufficient conditions can then either be used as a precondition for applying the translation or be used to insert conditional branches into the target code that cover those cases where the operational semantics would otherwise differ.

The general Fortran axiom is:

$$e < lb_F(A) \Rightarrow get_F(A, e) = error$$

Given map ϕ_2, this general axiom is translated:

$$\phi_2(e) < lb_F(A)$$
$$\Rightarrow get_C(\phi_2(A), \phi(e)) = error$$

The general C axioms include the a similar out of bounds axiom.

$$\phi_2(e) < lb_C(\phi_2(A))$$
$$\Rightarrow get_C(\phi_2(A), \phi_2(e)) = error$$

The general proof mirrors the specific proof. LTran needs to show $\theta_C(\psi_1(FC)) \vdash \phi_1(\theta_F(FC))$, which includes the following (where the ellipse represents the rest of the axioms for C):

$$[\phi_2(e) < lb_C(\phi_2(A)) \Rightarrow$$
$$get_C(\phi_2(A), \phi_2(e)) = error] \wedge \ldots \wedge$$
$$\vdash$$
$$[\phi_2(e) < lb_F(A) \Rightarrow$$
$$get_C(\phi_2(A), \phi_2(e)) = error]$$

For resolution refutation, the goal is negated and an attempt is made to derive the empty clause. (We replace '$get_C(\phi_2(A), \phi_2(e)) = error$' with 'exp' for readability):

1. $[\phi_2(e) < lb_C(\phi_2(A)) \Rightarrow exp] \wedge$
   $\neg[\phi_2(e) < lb_F(A) \Rightarrow exp]$
2. $[\neg(\phi_2(e) < lb_C(\phi_2(A))) \vee exp] \wedge$
   $\neg[\neg(\phi_2(e) < lb_F(A)) \vee exp]$
3. $[\neg(\phi_2(e) < lb_C(\phi_2(A))) \vee exp] \wedge$

$[(\phi_2(e) < lb_F(A)) \wedge \neg exp]$

4. $[\neg(\phi_2(e) < lb_C(\phi_2(A))) \wedge$
$(\phi_2(e) < lb_F(A)) \wedge \neg exp] \vee$
$[exp \wedge (\phi_2(e) < lb_F(A)) \wedge \neg exp]$
5. $[\neg(\phi_2(e) < lb_C(\phi_2(A))) \wedge (\phi_2(e) < lb_F(A)) \wedge$
$\neg exp]$

This will resolve to the empty clause when any subset of its conjuncts are false. Therefore, a derived sufficient condition for this proof to succeed is:

6. $[(\phi_2(e) < lb_C(\phi_2(A))) \vee (\phi_2(e) \geq lb_F(A))]$

This states that the translation is valid when the C index is lower than the lower bound of the C array or greater than or equal to the lower bound of the Fortran array.

6 Discussion

A preliminary version of LTran is underway, building on top of PROLOG-EBG. In this section we discuss user interface and theorem proving issues.

We anticipate three different classes of users for LTran. The first class would define the programming language theories, the second class would interactively define translation rules between languages, and the third class would use these translation rules to port programs. This paper has primarily addressed the interactive acquisition of translation rules from examples. LTran reduces the expertise needed for this second class of users by employing knowledge about programming language theories entered by the first class of users. Similarly, by storing semantic conditions needed for the valid application of translation rules, LTran reduces the expertise needed by the third class. We believe that LTran will make maintenance programmers substantially more productive. However, the primary benefit will be to improve the quality of ported programs by ensuring that translation rules are correct, while providing the freedom to meet non-functional requirements. This will enhance confidence in ported programs while at the same time reducing testing costs.

Proving implementation correctness is a major research issue with many practical applications. Although in the worst case proving implementation correctness is undecidable, there are many aspects which are amenable to current automatic theorem proving techniques. In our approach LTran can be used at different levels of abstraction with partial, decidable theories of the behavior of programming language constructs. There are many levels of abstraction for proving implementation correctness, depending on the granularity of the behavior which is being implemented. In some cases only high level invariants are critical. In other cases low level details such as the precision of arithmetic and the use of memory are important aspects of the behavior. The examples in this paper are based on high level invariants of abstract data types. As is shown by our examples, even this partial approach to deriving a set of axioms for the behavior of code fragments can provide considerable assistance to the maintenance programmer.

Acknowledgements

Thanks to Lee Blaine, Richard Jullig, Philip Laird, David Zimmerman, and the referees for reviewing this paper and providing helpful comments.

References

Blaine, L. and Goldberg, A. 1991. DTRE: A Semi-Automatic Transformation System. *Proceedings of IFIP TC2 Workshop on Constructing Programs from Specifications*, Asilomar, Pacific Grove, CA.

Burson, S., Kotik, G.B., and Markosian, L.Z. 1990. A Program Transformation Approach to Automating Software Re-engineering. *Proceedings of the Fourteenth International Computer Software and Applications Conference*, Washington, D.C.

Goguen, J. and Meseguer, J. 1982. Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. *Proceedings of ICALP*, Springer Verlag LNCS No. 140.

Kedar-Cabelli, S. T. and McCarty, L. T. 1987. Explanation-Based Generalization as Resolution Theorem Proving. *Proceedings of the Fourth International Machine Learning Workshop*, Irvine, CA.

Lowry, Michael R. 1990. Abstracting Domains with Hidden State. *Working Notes of the AGAA-90 Workshop*, Boston, MA.

Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. 1986. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1(1), 47-80.

Mitchell, T., Mahadevan, S. and Steinberg, L. 1985. LEAP: A Learning Apprentice for VSLI Design. *Proceedings of the Ninth IJCAI*, Los Angeles, CA.

Smith, R. G., Winston, H. A., Mitchell, T. M., and Buchanan, B. G. 1985. Representation and Uses of Explicit Justifications for Knowledge Base Refinement. *Proceedings of the Ninth IJCAI*, Los Angeles, CA.